Quarkus CDI

JEE Microservices

@ CGS IT – 2023

Version 1.0.5

Inhalt

- IoC – Inversion Of Control
- CDI – Context and Dependency Injection
- Quarkus - CDI Beans – Scopes
- Quarkus – Annotation Example
- Quarkus – CDI Limitations
- Quarkus – Dokumentation Links

Inversion of Control - IoC

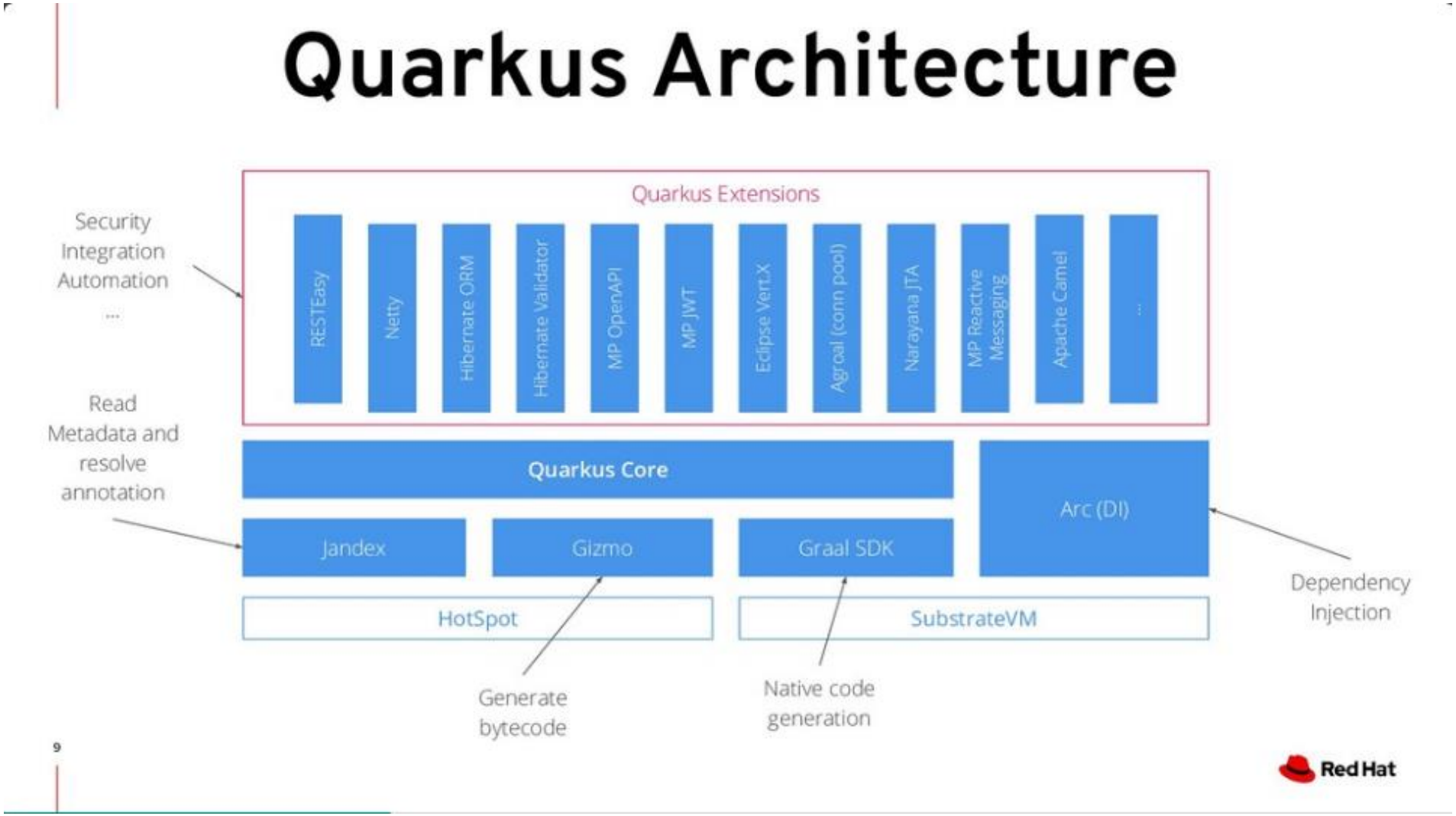
Steuerungsumkehr

- Nicht die Anwendung, sondern das Framework steuert den Programmfluss
- Das Framework
  - ruft Methoden "bei Bedarf" auf, z.B.
    - Eventhandler-Callbacks
    - Initialisierungsmethoden
    - Overrides
  - stellt bei Bedarf Objekte bereit
    - Instanzierung von Service-, Servlet, Bean-Objekten
    - löst Abhängigkeiten zwischen diesen Objekten auf

IoC – Inversion Of Control

- IoC in der Java EE
  - Kommt in diversen Technologien zum Einsatz:
    - Servlet
    - EJB
    - XML Webservices und REST Services
    - JMS
    - …
    - CDI (Context and Dependency Injection)

Quarkus - Architektur



Quarkus Container CDI

- Dependency injection in Quarkus is based on ArC which is a CDI-based dependency injection solution tailored for Quarkus' architecture.
- Quarkus only implements a subset of the CDI features and comes with non-standard features and specific APIS,

CDI – Context and Dependency Injection

- Dependency Injection
  - Vermindert Abhängigkeiten zwischen Objekten einer Java EE Anwendung
    - Lose Kopplung
    - Typsicherheit meist über Interfaces
  - Lebenszyklus wird vom Container gesteuert
  - Objekte werden vom Container bei Bedarf erzeugt und "injiziert"
  - Verwendbar in allen Komponenten, deren Lebenszyklus vom Container gesteuert wird
    - Servlet, Managed Bean, EJB, Webservice,
      REST Service, ...

Quarkus – CDI Beans

- A JavaBean in java is basically a POJO (Plain Old Java Object)
- Managed Bean is a Container Managed Bean
- A CDI **Bean** is a container-managed object that supports a set of basic services, such as injection of dependencies, lifecycle callbacks and interceptors.
- An application developer can focus on the business logic rather than finding out "where and how" to obtain a fully initialized component with all of its dependencies.

Quarkus CDI - Resolution

Quarkus CDI Annotations

Annotation	Description
`@Inject`	Identifies injectable constructors, methods, and fields
`@Qualifier`	Identifies qualifier annotations
`@ApplicationScoped`, `@SessionScoped`, `@RequestScoped`, `@Singleton`, `@Dependent`	Set of annotations defining the life cycle of a bean
`@Observes`	Identifies the event parameter of an observer method

CGS ITSolutions

Quarkus - CDI Beans - Scopes

Annotation	Description
@javax.enterprise.context.ApplicationScoped	A single bean instance is used for the application and shared among all injection points. The instance is created lazily, i.e. once a method is invoked upon the client proxy.
@javax.inject.Singleton	Just like @ApplicationScoped except that no client proxy is used. The instance is created when an injection point that resolves to a @Singleton bean is being injected.
@javax.enterprise.context.RequestScoped	The bean instance is associated with the current *request* (usually an HTTP request).
@javax.enterprise.context.Dependent	This is a pseudo-scope. The instances are not shared and every injection point spawns a new instance of the dependent bean. The lifecycle of dependent bean is bound to the bean injecting it - it will be created and destroyed along with the bean injecting it.
@javax.enterprise.context.SessionScoped	This scope is backed by a javax.servlet.http.HttpSession object. It's only available if the quarkus-undertow extension is used.

Quelle:
https://quarkus.io/guides/cdi#bean-scope-available

Quarkus Bean Discovery

- Bean classes that don’t have a bean defining annotation are not discovered. This behavior is defined by CDI.
- Beans with a Life Cycle Annotations are discovered

Quarkus – CDI - Inject

The `@Inject` annotation defines an injection point that is injected during bean instantiation. Injection can occur via three different mechanisms: property, setter, or constructor.

- Property Injection Annotates on a Property
- Constructor Injection Annotates on a Constructor Method
- Setter Injection Annotates on a Setter Method

CDI – Property Injection

```
@Path("/simplecdi")
public class CDISimpleResource {

    @Inject
    private SimpleCDIBean cdiBean;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String requestScope() {
        String new_value = cdiBean.echo("new value");
        return new_value;
    }}
```

Hier wird das benötigte CDI Bean direkt bei der Deklaration des Properties angegeben. Der CDI Container sucht das entsprechende Bean in der Liste der vorhanden Beans für das Inject.

CDI – Setter Injection

```
@RequestScoped
public class SetterInjection {

    private SimpleCDIBean cdiBean;

    @Inject
    public void setCdiBean(SimpleCDIBean cdiBean) {
        this.cdiBean = cdiBean;
    }

    public String echo(String input) {
        log.info("SetterInjection");
        return cdiBean.echo(input);
    }}
```

CDI – Constructor Injection

```
@RequestScoped
public class ConstructorInjection {

    SimpleCDIBean cdiBean;

    @Inject
    public ConstructorInjection(SimpleCDIBean cdiBean) {
        this.cdiBean = cdiBean;
    }

    public String echo(String input) {
        log.info("SetterInjection");
        return cdiBean.echo(input);
    }
}
```

CDI – Quarkus Default Inject Keyword

- Das @Default Schlüsselwort kann verwendet werden, um die Default Implementierung eines CDI Beans zu „verlangen“.

```
@Path("/simplecdi")
public class CDISimpleResource {

    @Inject
    private SimpleCDIBean cdiBean;

    @Inject
    @Default
    private ConstructorInjection cinj;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String requestScope() {
```

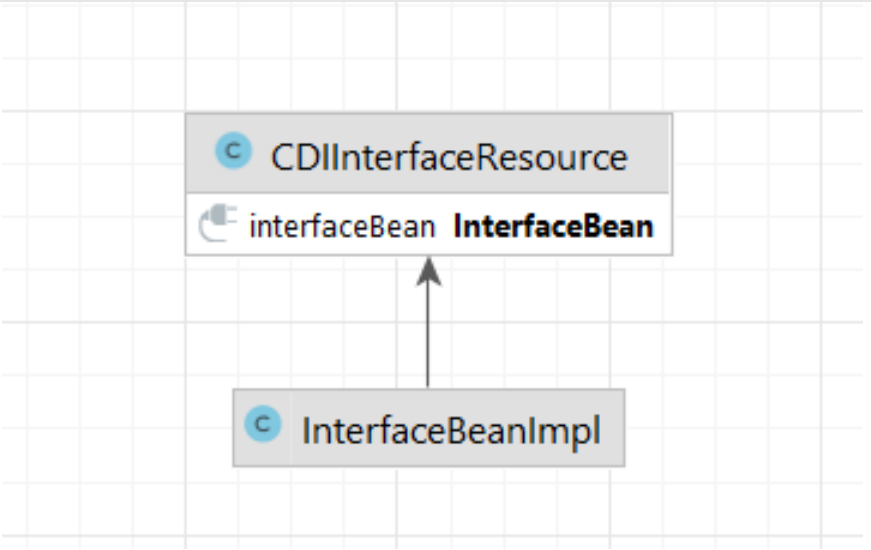
CDI – Inject Interface & Use Implementation

- Der Injection Point kann auch ein Interface sein.
- Mindestens 1 Implementierung muss vorhanden sein.
- Injected wird die vorhandene Implementierung als Default.

```
@Path("/simplecdi_interface")
public class CDIInterfaceResource {

    @Inject
    InterfaceBean interfaceBean;

…
}
```



CGS ITSolutions

Quarkus – CDI Qualifiers

Um bei mehreren vorhandenen Implementierungen eine entsprechende Implementierung auswählen zu können, kann dafür ein eigener CDI Qualifier implementiert werden.

```
import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface QualifyA {
}
```

```
@QualifyB
@ApplicationScoped
public class QBeanImplB implements QBean, Serializable {

   @Inject
   Logger log;

   @Override
   public String echo(String input) {
...
}}
```

CDI Qualifier Resolve Error

```
Resulted in: jakarta.enterprise.inject.UnsatisfiedResolutionException: Unsatisfied
dependency for type at.cgsit.jeemicro.cdi.qualify.QBean and qualifiers [@Default]
    - java member: at.cgsit.jeemicro.resource.cdi.CDIQualifyResource#interfaceBean
    - declared on CLASS bean [types=[at.cgsit.jeemicro.resource.cdi.CDIQualifyResource,
java.lang.Object], qualifiers=[@Default, @Any],
target=at.cgsit.jeemicro.resource.cdi.CDIQualifyResource]
    The following beans match by type, but none have matching qualifiers:
- Bean [class=at.cgsit.jeemicro.cdi.qualify.QBeanImplB, qualifiers=[@Any, @QualifyB]]
- Bean [class=at.cgsit.jeemicro.cdi.qualify.QBeanImplA, qualifiers=[@Any, @QualifyA]]
```

CGS ITSolutions

CDI Qualifier Verwendung

- Erst durch die Qualifizierung bei der Verwendung des Beans wird diese Mehrdeutigkeit aufgelöst

```
@Path("/simplecdi_qualify")
public class CDIQualifyResource {

   @Inject
   @QualifyB
   QBean interfaceBean;
```

CDI Alternative Beans (Mock)

```
@Alternative
@ApplicationScoped
public class AlternativeBeanImplDummy
implements AlternativeBeanInterface,
Serializable {

    @Inject
    Logger log;

    @Override
    public String echo(String input) {
… }}
```

- Alternative Beans können mittels der Annotation @Alternative zur Verfügung gestellt werden.
- Diese Beans können mittels application Configuration Property aktiviert werden.
- Und ersetzten dann die Default Implementierung

```
# cdi alternatives
quarkus.arc.selected-alternatives=at.cgsit.jeemicro.cdi.alternatives.mock.*
```

CDI – Producer Methoden

- @Produces erlaubt es die Erzeugung von CDI Beans selbst zu implementieren.
- Dabei kann auch der Scope der Beans definiert werden.
- Hier in Kombination mit einem Konfigurations Propery.

```
public class PBProducer {

   @ConfigProperty(name = "at.cgs.training.produceBean",
defaultValue = "a")
   String beantoProduce;

   @Produces
   @RequestScoped
   PBInterface producePB() {
      log.info("producer called");
      if( "a".equalsIgnoreCase(beantoProduce)){
         return new PBImplA();       }
      return new PBImplB();    }}
```

CDI Producer Konfiguration Beispiel

```
# wechseln der Implementierung von A auf B via Producer Methode und diesem
Konfigurationsparameter
at.cgs.training.produceBean=b
# at.cgs.training.produceBean=a
```

CDI Lifecycle Callbacks

1. @PostConstruct:
   This callback is invoked before the bean instance is put into service. It is safe to perform some initialization here.
2. @PreDestroy:
   This callback is invoked before the bean instance is destroyed. It is safe to perform some cleanup tasks here.

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;

@ApplicationScoped
public class Translator {

    @PostConstruct 1
    void init() {
       // ...
    }

    @PreDestroy 2
    void destroy() {
      // ...
    }
}
```

```
# cdi
# quarkus.arc.selected-alternatives=org.acme.beans.mock.*
```

CGS ITSolutions

CDI Interceptoren

- Ein Interceptor gibt die Möglichkeit vor für CDI Bean Methoden vor und nach dem Aufruf der jeweiligen Ziel Bean Methode eigenen Source Code auszuführen.
- Dies kann zum Beispiel für eigenes Logging (wie in diesem Beispiel gezeigt) genutzt werden
- Priority(2020)
  Priority enables the interceptor and affects the interceptor ordering. Interceptors with smaller priority values are called first.

```
@Logged
@Priority(2020)
@Interceptor
public class LoggingInterceptor {

   @Inject
   Logger logger;

   @AroundInvoke
   Object logInvocation(
InvocationContext context) throws Exception {

      logger.info("object before");
      Object ret = context.proceed();
      logger.info("object after");
      return ret;
   }
}
```

CGS ITSolutions

CDI – Annotation für Interceptor Binding

- Dafür muss eine eigene Annotation programmiert werden.
- Diese Annotation implementiert ein jakarta Interceptor Binding.
- Das Binding kann dann auf Klassen, Methoden oder Construktor Scope im Ziel verwendet werden

```
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD,
ElementType.CONSTRUCTOR})
@Inherited
public @interface Logged {

}
```

CGS ITSolutions

CDI Annotation Verwendung

- Die Verwendung sieht für ein klassenweites Logging folgend aus

```
@RequestScoped
@Logged
public class RSBeanInterceptedExample {

   public String echoReverse(String input) {

         …
         return reverse.toString().toUpperCase(Locale.ROOT);
   }

   public String echoReverse2(String input) {
      String reverse = echoReverse(input);
      return echoReverse(reverse);
   }

}
```

CDI – Priorities Table

`@javax.interceptor.Interceptor.Priority` takes an integer that can be any value. The rule is that values with a lower priority are called first. The `javax.interceptor.Interceptor` annotation defines the following set of constants:

- `PLATFORM_BEFORE` = 0: Start of range for early interceptors defined by the platform,
- `LIBRARY_BEFORE` = 1000: Start of range for early interceptors defined by extension libraries,
- `APPLICATION` = 2000: Start of range for interceptors defined by applications,
- `LIBRARY_AFTER` = 3000: Start of range for late interceptors defined by extension libraries, and
- `PLATFORM_AFTER` = 4000: Start of range for late interceptors defined by the platform.

CDI – Request Scoped Bean

- Ein RequestScoped Bean wird immer dann neu erzeugt, wenn ein neuer http Request verarbeitet wird.
- Wird also dieses Bean in einer JaxRS Service Resource verwendet, wird für jeden Request eine neue eigene Bean Objektinstanz instanziiert.

```
@RequestScoped
public class RSBean {

   @Inject
   Logger log;

   private String requestScopedMessage = "default";

   @PostConstruct
   public void postConstruct() {
      log.info("postConstruct called: " + LocalDateTime.now());
   }

   public String getRequestScopedMessage() {
      return requestScopedMessage;    }

   public void setRequestScopedMessage(String requestScopedMessage)
{
      this.requestScopedMessage = requestScopedMessage;
   } }
```

CDI Request Scope Beispiel

- Initialisierung zwischen dein einzelnen Requests mit neuen Objekten und Post-Construct-Aufrufen.

resteasy-reactive, resteasy-reactive-jackson, security, security-jpa, smallrye-context-propagation, smallrye-openapi, swagger-ui, vertx]
TEE: 14:04:53 INFO [io.qu.de.de.RuntimeUpdatesProcessor] (vert.x-worker-thread-1) Live reload total time: 2.275s
TEE: 14:04:53 INFO [at.cg.je.cd.re.RSBean] (executor-thread-1) postConstruct called: 2023-05-30T14:04:53.760867100
TEE: 14:05:12 INFO [at.cg.je.cd.re.RSBean] (executor-thread-1) postConstruct called: 2023-05-30T14:05:12.467123100
TEE: 14:05:13 INFO [at.cg.je.cd.re.RSBean] (executor-thread-1) postConstruct called: 2023-05-30T14:05:13.947806500

CDI – Application Scoped Bean

- Das Application Scoped Bean wird nur Einmal pro Quarkus Instanz erzeugt.

```
@ApplicationScoped
public class ApplicationScopeBean {

   public Integer counter = 0;

   public Integer getCounter() {
      this.counter++;
      return counter;
   }
}
```

CDI - Events

- Ein CDI Bean kann ein Event erzeugen (fire).
- Dieses Event wird mittels Event Injection Annotation deklariert.
- Die eigene Event Klasse “SpecialEvent“ kann nach Bedarf implementiert werden

```
@Singleton
public class SpecialService {

   @Inject
   Event<SpecialEvent> event;

   public void doSomething() {
      event.fire(new SpecialEvent("Special Event"));
      log.info("SpecialService called AFTER event");
   }
`
public class SpecialEvent {

   private String message;

   public SpecialEvent(String message) {
      this.message = message;   }

   public String getMessage() {
      return message;
   }}
```

```
SpecialService called before event.fire
SpecialEventListener called: Special Event
SpecialService called AFTER event
```

CGS ITSolutions

CDI – Events Consumer

- Das Event kann durch die Annotation @Observes <EventType> empfangen werden

```
@ApplicationScoped
public class SpecialEventListener {

   @Inject
   Logger log;

   void onSpecialEventCompleted(@Observes SpecialEvent
event) {
      log.info("SpecialEventListener called: " +
event.getMessage());
   }

}
```

CDI - Build Profiles

By default Quarkus comes with three profiles:

1. dev – Activated when in development mode: mvn quarkus:dev
2. test – Activated when running tests: mvn test
3. prod – The default profile when not running in development or test mode
4. @UnlessBuildProfile(“prod”) ermöglicht das Umdrehen dieses Konzepts.

```
@Dependent
public class TracerConfiguration {

   @Produces
   @IfBuildProfile("prod")
   public Tracer realTracer() {
      return new TracerImplTwo();
   }

   @Produces
   @DefaultBean
   public Tracer noopTracer() {
      return new TracerImplTwo();
   }
}
```

CGS ITSolutions

Quarkus CDI Limitations

- @ConversationScoped is not supported
- Portable Extensions are not supported
- BeanManager - only the following methods are implemented:
  - getBeans(), createCreationalContext(), getReference(), getInjectableReference() , resolve(),
  - getContext(), fireEvent(), getEvent()
  - and createInstance()
- Specialization is not supported
- beans.xml descriptor content is ignored
- Passivation and passivating scopes are not supported
- Interceptor methods on superclasses are not implemented yet

Quarkus – Dokumentation Links

- Quarkus Logging

https://quarkus.io/guides/logging

- Jboss Logging
  https://docs.jboss.org/seam/3/latest/reference/en-US/html/solder-logging.html

Danke für Ihre Aufmerksamkeit

CGS ITSolutions